Hindawi Publishing Corporation
VLSI Design
Volume 2012, Article ID 793196, 11 pages
doi:10.1155/2012/793196

Research Article

An Empirical Investigation on System and Statement Level Parallelism Strategies for Accelerating Scatter Search Using Handel-C and Impulse-C

M. Walton, O. Ahmed, G. Grewal, and S. Areibi

School of Engineering and Computer Science, University of Guelph, Guelph, ON, Canada N1G 2W1

Correspondence should be addressed to S. Areibi, sareibi@uoguelph.ca

Received 4 March 2011; Revised 4 August 2011; Accepted 1 October 2011

Academic Editor: Gregory Peterson

Scatter Search is an effective and established population-based metaheuristic that has been used to solve a variety of hard optimization problems. However, the time required to find high-quality solutions can become prohibitive as problem sizes grow. In this paper, we present a hardware implementation of Scatter Search on a *field-programmable gate array (FPGA)*. Our objective is to improve the run time of Scatter Search by exploiting the potentially massive performance benefits that are available through the native parallelism in hardware. When implementing Scatter Search we employ two different *high-level languages (HLLs)*: Handel-C and Impulse-C. Our empirical results show that by effectively exploiting source-code optimizations, data parallelism, and pipelining, a 28x speed up over software can be achieved.

1. Introduction

Most optimization problems found in real-world applications are NP-hard [1]. Consequently, exact methods (which provide optimal solutions) are often abandoned in lieu of metaheuristics, which seek to provide approximate solutions in a reasonable amount of time. One of the most effective and well-known metaheuristics is *Scatter Search*. First proposed by Glover [2], Scatter Search is a population-based search technique that has been successfully applied to a variety of problems, primarily in the areas of combinatorial optimization [3] and machine learning [4]. Although Scatter Search can significantly reduce the temporal complexity of the search process, the former remains time-consuming for real-world problems. Given the abundance of parallel computers available today, exploiting parallelism appears to be the most natural way to speed up the search for approximate solutions. Recently, several parallel Scatter Search implementations have been proposed [4, 5]. These implementations typically seek to speed up Scatter Search, with respect to its sequential counterpart, by parallelizing the Scatter Search procedure to run either in a distributed or shared-memory environment. Moreover, some of these implementations seek to find improved solutions or attempt to solve larger problem instances. The latter is possible since parallel implementations often search the problem search space in a different manner compared to their sequential counterparts [6]. The conclusions reached in these studies are that different parallel strategies do have an effect on the quality of solutions found, as well as the overall speed up that can be achieved. In all cases, however, the speed ups reported were less than one order of magnitude over the sequential version of the same Scatter Search procedure. In this paper, we seek to further improve the performance and accelerate Scatter Search by implementing the metaheuristic directly on reconfigurable computing systems (RCS) in the form of field-programmable gate arrays (FPGAs).

1.1. RCS and Electronic System Level Design. As modern FPGAs contain close to one million programmable logic blocks, they exhibit substantial opportunities for performing parallel computation. Moreover, FPGAs provide a natural environment in which pipelining and data parallelism strategies can also be employed to speed up the Scatter Search procedure. Recently, FPGAs have been used to accelerate other

time-consuming algorithms, including Genetic Algorithms [7], Memetic Algorithms [8], and Neural Networks [9]. However, we are unaware of any attempt to use FPGA-based acceleration to speed up Scatter Search. When mapping any algorithm onto an FPGA for the purposes of acceleration, the first step requires encoding the algorithm using a suitable language for hardware synthesis. The two most common languages for designing hardware are VHDL [10] and Verilog [11]. Both of these languages describe hardware at the register transfer level (RTL). Consequently, they force the designer to be occupied with the structural details of the actual hardware. More recently, higher-level languages, such as Handel-C [12], Impulse-C [13], and Catapult-C [14], have been proposed for the purposes of hardware acceleration of software applications. The primary aim of these languages is to enable designers to focus their attention on the algorithm to be designed rather than the circuit to be built. Most importantly, all of these languages require the designer to have minimal hardware knowledge, thus making the languages suitable for use by a wider audience of software developers interested in accelerating their algorithms by implementing them directly in hardware. In practice, the results of software-to-hardware compilation from C-like descriptions will not (usually) equal the performance of hand-coded VHDL/Verilog, but the turnaround time to get those first results may be an order of magnitude faster.

1.2. Overall Approach. In this paper we build upon our earlier work in the area of reconfigurable hardware systems [15, 16] and propose an architecture for a Scatter Search engine that employs a combination of statement-level and system-level optimization to achieve significant speed ups. To do this, we implement and evaluate over thirty different Scatter Search implementations created using Handel-C and Impulse-C. The syntax of these languages is based on the syntax for ANSI C; however, instructions are implemented as logic gates, and constructs exist to allow multiple instructions to execute in parallel. An important consideration when using these languages is that poorly written code will frequently result in a poor hardware implementation, that is, a hardware implementation that requires excessive numbers of clock cycles and/or exhibits low clock frequency. Therefore, designers are encouraged to optimize their original source code by applying both statement-level and system-level optimization techniques. The former consists of numerous fine-grained optimizations that operate at the level of individual statements and which may, or may not, be controlled by the compiler. The latter includes coarse-grained parallel coding techniques, like pipelining and data parallelism. Both forms of optimization are critical to obtaining a high-performance hardware implementation. However, the sheer volume of these optimizations leads to a whole design space of potential solutions, all based on different combinations of optimizations.

1.3. Contributions and Organization. The main contributions of our work include the following:

(1) proposing an architecture for a Scatter Search engine that employs a combination of statement-level and system-level parallelism to achieve a speed up of 28x over software,

(2) performing a statistical study to measure the effects of statement-level parallelism in Handel-C and Impulse-C on CPU cycles and clock frequency, and,

(3) comparing the effectiveness of Handel-C to Impulse-C with regard to accelerating Scatter Search.

The remainder of this paper is organized as follows. Section 2 describes the application of Scatter Search to the 0-1 knapsack problem and discusses high-level languages. The conversion of Scatter Search to Handel-C and Impulse-C and its initial mapping onto the FPGA are discussed in Section 3. Results based on fine-grain parallelism are introduced in Section 4. Further improvements to the Scatter Search algorithm, based on data parallelism and pipelining, are discussed in Section 5. Finally, we provide our conclusions and future works in Section 6.

2. Background

The 0-1 knapsack problem [17] is a well-known combinatorial optimization problem that can be formally stated as follows:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i, \\ \text{s.t.,} \quad & \sum_{i=1}^{n} w_i x_i \leq C, \end{aligned} \tag{1}$$

where v_i is the value of item i, w_i is the weight of item i, C is the capacity of the knapsack, and x is a 0-1 variable; if x_i is 1, item i is selected to appear in the knapsack.

The Scatter Search algorithm that we employ for solving the 0-1 knapsack problem is based on the Scatter Search template first described by Glover [2].

As shown in Figure 1, the Scatter Search template consists of two stages.

(1) *Stage I* is responsible for generating a diverse set of high-quality solutions for the problem being solved. This is accomplished using both Diversification and Improvement methods.

   (a) The *diversification method* is responsible for producing solutions that differ from each other in significant ways and that yield productive alternatives in the context of the problem being solved.

   (b) The *improvement method* is responsible for improving the quality of the initial solutions, typically by using a form of local search. The initial set of solutions is created by iteratively generating new solutions by invoking the diversification method followed by the improvement method. The actual number of initial solutions

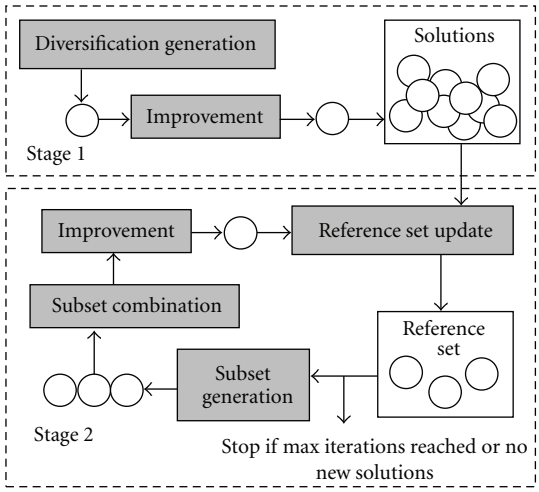

Figure 1: Scatter Search template.

generated is typically one hundred (or ten times the size of the actual number that will eventually enter the reference set).

(2) *Stage II* begins with the application of the reference set update method followed by a subset Generation method.

   (a) The *reference set update method* typically accompanies each application of the improvement method and is used, on the first iteration, to designate a subset of initial solutions to the reference set. The primary purpose of creating and maintaining a reference set is to have a current collection of diverse, high-quality solutions at hand for generating further solutions through a combination strategy.

   (b) The *subset generation method* is then used to systematically group the solutions in the reference set into subsets of size 2 (or more) individuals. These individuals are chosen to produce points both inside and outside the convex regions spanned by the reference solutions.

   (c) Once determined, the *solution combination method* is used to transform each subset of solutions produced by the subset generation method into one combined solution.

   (d) As there is no guarantee that the new solutions are of high quality, the *improvement method* is used to improve the quality of the new solutions. The reference set update method is then applied once more. At this step, a new improved solution can replace an existing solution in the reference set, but only if it has superior quality or diversity.

(3) The entire process in the second stage then repeats until a maximum number of iterations is performed, or until the reference set does not change. The "best" solution in the reference set is reported as the final solution.

2.1. High-Level Languages (HLLs). Hardware description languages (HDLs) are tailored specifically to hardware design. Because of this, they provide a flexible and powerful way to generate efficient logic. However, this tailoring makes them unfamiliar territory for people outside the hardware design field. In order to communicate hardware design to a more general audience, a number of tools are emerging to support the use of other high-level programming languages (primarily C and C++) as HDLs. Modern software-to-hardware tools such as Catapult-C, Impulse-C, Mitrion, and Handel-C support this type of application development by allowing designers to describe applications in a traditional software-oriented environment.

2.1.1. Handel-C. Handel-C is a high-level programming language that allows programmers with little hardware expertise to easily convert their algorithms into hardware. Based on a subset of ANSI-C, programs written in Handel-C are implicitly sequential with instructions being executed in the order in which they appear. Handel-C provides direct control over parallelism by allowing the application developer to describe such things as clocks and resets, and by specifying (through the use of "par" and other statements) how C code is parallelized when implemented as hardware.

2.1.2. Impulse-C. Impulse-C is also a high-level programming language that supports the development of highly parallel, mixed hardware/software algorithms and applications using processes and streams. At the heart of the Impulse-C tools is a software-to-hardware compiler that converts processes to functionally equivalent hardware descriptions, as well as generating the required interfaces implementing the specified streams, memories, and signals. Expressing parallelism at the system level within Impulse-C is similar to other C-based languages, including Celoxica Handel-C and System-C. However, Impulse-C represents an untimed method of expressing applications.

2.1.3. Handel-C versus Impulse-C. There are several differences between Handel-C and Impulse-C [18] that are worth mentioning.

(1) The specification of clock timing and statement-level parallelism is the most fundamental difference between Impulse-C and Handel-C. Handel-C requires that clock boundaries be clearly specified by the programmer, while Impulse-C does not.

(2) Impulse-C [19] is designed for software programmers who are familiar with C programming, but who may not be familiar with hardware design concepts. Impulse-C, therefore, does not require the specification of clocks, resets or statement-level parallelism. Instead, Impulse-C assumes that the compiler, directed (several pragmas are inserted by the designer for loop unrolling and pipelining) by the designer,

will automatically extract parallel behaviors from a given application.

(3) The approach in Impulse-C mainly focuses on mapping algorithms to a mixed FPGA/processor (hardware/software codesign) system with the goal of creating hardware implementations of processes that optionally communicate with software processes residing on an embedded microprocessor.

2.1.4. Code Optimization. While Handel-C and Impulse-C automate much of the task of converting programs into hardware, they do not optimize the original source code. Thus, poorly written source code may be synthesized into poor hardware implementations that require a significant number of clock cycles or which have low clock frequencies. Consequently, these languages encourage developers to manually restructure their code with the aim of optimizing the resulting hardware. In general, these manual optimizations have the potential to reduce clock cycles, increase clock frequency, or perhaps both. However, the opportunities for code restructuring are vast, leading to a whole design space of potential solutions, all based on different optimizations. Moreover, there is, to date, no a priori evidence to suggest which combinations of optimizations may be helpful and which may be detrimental when seeking to optimize hardware. Nor are the sizes of performance gains/losses that a developer might expect when using different combinations of optimizations known. Therefore, to answer these questions, we propose to perform a full-factorial design that will allow us to consider and test interactions between different combinations of language-level transformations and optimizations.

3. Methodology

Our hardware implementation of Scatter Search is based on the actual implementation provided by Laguna and Mart [3]. Like Laguna and Mart, we choose to deterministically generate an initial population of size 100, which is later culled to 10 in order to form the reference set. We also limit our combination method to working with subsets of solutions of size 2. The only modification that we chose to make to the implementation in [3] was to replace the dynamic memory allocation with references to a global data structure stored in RAM (on the FPGA). This change was necessary as Handel-C and Impulse-C do not support dynamic memory allocation. Otherwise, all of the Scatter Search methods were implemented as described in [3].

3.1. Handel-C Optimization. The advantage of performing a full-factorial design is that the interaction between all factors can be studied. However, the number of runs goes up exponentially as additional factors are added. Experiments with many factors can quickly become unwieldy and costly to execute. Therefore, to keep the size of the full-factorial design manageable, we have chosen to organize the various code-restructuring optimizations available in Handel-C into five logical groups based on their similarity. For larger designs, or to study higher numbers of factors and interactions, fractional-factorial designs can be used to reduce the number of runs by evaluating only a subset of all possible combinations of factors. However, it should be noted that although these designs are very cost effective, the study of interactions between factors is limited.

Table 1 identifies the five optimization groups (labeled A through E), each group's constituent optimizations, and the potential effects of these optimizations on clock frequency and the number of clock cycles. In general, these optimizations seek to improve the run time performance of the synthesized hardware by increasing the clock frequency of the design and/or by reducing the total number of required clock cycles. With regard to the former, certain operations in Handel-C can combine together to form deep logic. This deep logic results in long delays through critical paths in the design, which adversely affects performance. Many of the previous optimizations seek to reduce deep logic in order to increase clock speed. Other optimizations seek to restructure code or execute sections of code in parallel to reducing the total number of required clock cycles.

Due to space limitations, it is not possible to provide detailed descriptions of each code-restructuring optimization in each group. However, group A consists of a variety of loop-based optimizations that seek to reduce the number of clock cycles required to perform a loop by parallelizing the logic required for the loop body and loop test, when possible. In particular, the first optimization in group A seeks to replace *for* loops with *while* loops; the second seeks to employ unsigned overflow to efficiently terminate a loop; the third seeks to reduce clock cycles by moving the first (or last) iteration of a loop outside the loop and parallelizing its execution with existing statements in the prologue (epilogue); the fourth optimization seeks to reduce the logic required to implement the loop by replacing *while* loops with *do* loops, when possible.

Group B contains optimizations that seek to reduce deep logic by removing complex expressions from conditional tests and by restructuring deeply nested conditional statements. More specifically, the first optimization in group B seeks to break up deep logic by registering the results of conditionals in Boolean variables; the second optimization inserts delay statements into the (unused) else blocks of *if* statements to improve clock frequency and avoid combinational cycles; the third optimization seeks to reorganize *if-else* statements into more (logic) efficient ternary expressions.

The first optimization in group C seeks to improve clock frequency by registering (caching) data retrieved from RAMs closer to where the data is actually used in the circuit. The second optimization seeks to (potentially) reduce the length of the critical path by creating unique identifiers (registers) for all uses of a common variable.

In group D, the first optimization seeks to reduce deep logic by simplifying complex expressions into series of simpler computations. The second optimization seeks to reduce clock cycles by moving loop-invariant statements outside of the loop in which they appear.

Table 1: Handel-C optimization summary.

Group	Strategy	Frequency	Cycles
Loops (A)	Replace *for*	No effect	Decrease
	Overflow	Increase	No effect
	dec. Iterations	Decrease	Necrease
	use do...while	Increase	No effect
Conditionals (B)	Simplify	Increase	Increase
	Delay	Increase	Increase
	Reorganize	Increase	No effect
Registering (C)	RAM access	Increase	Increase
	Unique identifier	Increase	No effect
Expression (D)	Simplify	Increase	Increase
	Movement	No effect	Decrease
Parallel (E)	Par{}	No effect	Decrease

Finally, group E contains the par{} construct described earlier, which can be used to reduce clock cycles by executing statements simultaneously on separate hardware units. More detailed information regarding the optimizations employed can be found in [20].

3.2. Impulse-C Optimization. Code restructuring optimization available in Impulse-C can be organized into two main groups, as seen in Table 2. Initial optimization is usually performed by the compiler automatically and, therefore, will not be discussed further. Loop unrolling is used generally in system design to improve throughput and optimize critical parts of the system by duplicating hardware components. The penalty of this approach will be an increase in area since unrolling a loop tends to duplicate hardware. Also, as the number of loops unrolled increases, it is intuitive that the longest path delay of the circuit increases as more components are connected in sequence and fewer resources are shared iteratively. Consequently, the maximum clock frequency decreases; but, as the maximum clock frequency decreases, the total number of cycles needed is also reduced.

Pipelining, on the other hand, can be applied to loops that require two or more cycles per iteration and that do not have dependencies between iterations. The goal of pipelining is to execute multiple iterations of a loop in parallel and thus improve throughput.

(1) *Initial optimization:* this phase performs various common optimizations, such as constant folding and dead-code elimination, just to name a few.

(2) *Loop unrolling:* in this phase whenever the compiler encounters the *UNROLL pragma* it performs a corresponding expansion of loops into equivalent parallel statements.

(3) *Pipelining:* this is one of the most significant types of optimizations to consider when generating hardware. To generate a pipeline using Impulse-C, a *CO PIPELINE pragma* must be inserted inside the main loop.

4. Experimental Framework

To gauge the influence that the previous groups of optimizations have on the performance of Scatter Search when implemented using Impulse-C and Handel-C, an extensive experimental analysis is performed, where all possible combinations of optimizations are considered.

Each version of Scatter Search is developed by first converting the original code proposed by Laguna and Mart to Handel-C and Impulse-C, respectively. We refer to this code as the *base*. Then, optimizations are manually added to the base code for each version. Due to the deterministic nature of Scatter Search and the fact that none of the optimizations change the functionality of the algorithm, all hardware versions produce the same solutions for the same problem instances. Therefore, when comparing the hardware versions among themselves, it is not necessary to compare the versions with respect to solution quality. Rather, versions need only be compared with respect to run time. In reality, the run time of each algorithm can only be known once the circuit has been placed and routed on the FPGA. Following placement and routing, the clock rate (frequency) of the design can be determined from the reciprocal of the clock period. The clock rate can then be multiplied by the number of clock cycles required to execute the design to determine the actual run time of the algorithm mapped onto the FPGA.

4.1. Benchmarks. All versions of Scatter Search were tested with 10 benchmarks generated according to the procedure outlined in [21]. Each benchmark corresponds to a knapsack problem of size 50 items. This problem size was chosen due to the large number of hardware versions and a desire to avoid excessive placement and routing times associated with larger problem instances. All 10 benchmarks were applied to each version and the (arithmetic) mean run times recorded. These mean run times were then used to calculate the speed up achieved compared to software, where the speed up is defined as the ratio of the mean time required to run the algorithm in software over the mean time required to run the algorithm in hardware.

4.2. Technical Issues. The design environment employed in this work is based on tools provided by Xilinx [22], Mentor Graphics [20], and Impulse [13]. Each of the Handel-C versions was developed using the DK Design Suite, Version 5.0. The synthesized circuits were mapped onto the FPGA using Xilinx ISE Version 11. The targeted FPGA was a Xilinx Virtex-5 LX. When comparing with software, Scatter Search was implemented in the C language and run on an Intel Core2 Duo machine with a 1.8 GHz CPU and 1 GB of main memory, running Ubuntu 9.04.

4.3. Results for Handel-C. As discussed earlier, five groups of optimizations, A, B, C, D, and E, were examined and implemented for Handel-C. Thus, a total of 31 different hardware versions are considered in this section.

Table 3 shows the mean run times of the 31 versions. Also included in Table 3 are the run times of the base code

Table 2: Impulse optimization summary.

Group	Strategy	Frequency	Cycles
Loop Unrolling	Used to improve throughput The penalty is an increase in area	Decrease	Decrease
Pipelining	Used to improve throughput by reducing the number of instruction cycles	Increase	Increase

(i.e., Scatter Search with no Handel-C optimizations present) and the run time of Scatter Search in software. Before attempting to draw conclusions from the rankings in Table 3 with regard to the overall effectiveness of various combinations of optimizations, an analysis of variance (ANOVA [23]) was performed on the data. ANOVA is a statistical tool that can be used to study variability in data, like those reported in Table 3. More specifically, it is a method of analyzing the variance to which a response is subject into its various components, corresponding to the sources of variation that can be identified [24]. Working with the data in Table 3, all 32 versions (including the base) were investigated for any overall differences between them. There were statistically significant differences identified so then pair wise comparisons of all 32 algorithms were made to find where specific differences occurred. This analysis was corrected using a Bonferroni correction for multiple comparisons [25]. We chose a significance level of 0.01 to identify statistical significance. The ANOVA [23] results were based on a model controlled for the benchmarks. Our analysis revealed the following.

(i) A straightforward mapping of Scatter Search into Handel-C does not result in a speed up with respect to software; this is witnessed by the base version running 14% slower than software.

(ii) A speed up over software can be obtained by manually applying source-code optimizations. However, these optimizations must be applied judiciously; 23 of the 31 versions managed to obtain a speed up compared to software, but 9 versions resulted in a slowdown.

(iii) Among individual optimizations, group C, which seeks to register (cache) data accessed from RAM close to where it is required in the actual circuit, provides the largest speed up. However, there is no statistical difference between groups C and E. What is important about optimization C is its potential to improve clock frequency by reducing the delay when accessing data directly in RAM. The hardware version based only on C is able to increase the clock rate (compared with the base) by 207%. This increase in clock frequency, however, is not free, as 20% more clock cycles (compared with the base) are required as all reads/writes from/to memory must be cached. In general, this optimization is likely to be effective when the Handel-C code contains lots of memory accesses.

(iv) The largest speed ups are obtained by two combinations of optimizations: DE (97%) and ABDE (90%). However, there is no statistical difference between these two. Thus, given that the latter required more effort on the part of the developer to implement, we consider DE to be the better choice. The overall effectiveness of this combination of optimizations (i.e., DE) can be attributed to the fact that both optimizations combine to reduce the total number of clock cycles (relative to the base) by 47%.

When applied individually, optimizations D and E result in a speed up of 5% and 38%, respectively. However, when both optimizations are used together, abundant opportunities exist for the restructured and simplified statements produced by optimization D to be performed in parallel with other statements. This leads to a significant reduction in the total number of clock cycles (43% compared with optimization D and 17% compared with optimization E) as well as a further improvement in clock frequency (7% compared with optimization D and 18% compared with optimization E).

(v) Interestingly, using all 5 groups of optimizations (ABCDE) does not result in the largest speed up (74%); however, it does require the most effort on the part of the developer to implement.

The FPGA resource utilities of the fastest implementation (DE) along with the base implementation are shown in Table 4. Note that the FPGA used to implement both designs contains 67,584 slices, 288 block RAMs, and 96 DSP blocks. Interestingly, it can be seen from Table 4 that both hardware implementations employ roughly the same amount of resources. However, across all 31 implementations we found that the addition of group C optimizations resulted in a 17% increase in slice flip flops and a 24% increase in slice 4 LUTs, while the addition of group B and group E optimizations resulted in a 3% increase and 4% decrease in the number of flip flops, respectively.

4.4. Results for Impulse-C. Table 5 shows the resources consumed by the base version along with the independent optimized techniques applied. Also in Table 5, the maximum operating frequency, clock cycles, and average time are reported.

It is interesting to observe the increase in FPGA resources required by both the "loop unrolling" versus the "pipelining" strategies. As discussed earlier the "loop unrolling" strategy

Table 3: Handel-C results for factorial design.

Optimization version (with respect to base)	Clock freq. (MHz)	Average cycles	Average run time (s)	Speed up (with respect to software)
Software	—	—	0.48278	1.00
Handel-C Base (No Opt)	15.36	8583039	0.55869	0.86
A	13.6	6980327	0.51318	0.94
B	18.71	13851615	0.7402	0.65
C	31.89	10788058	0.33826	1.43
D	17.22	7883208	0.45767	1.05
E	15.49	5412596	0.34931	1.38
AB	18.21	6509286	0.35753	1.35
AC	13.99	8971769	0.6411	0.75
AD	16.32	5435436	0.33296	1.45
AE	15.12	5258982	0.34774	1.39
BC	28.4	17479877	0.61554	0.78
BD	20.28	13222446	0.65208	0.74
BE	18.28	9415528	0.51512	0.94
CD	32.34	10048769	0.31075	1.55
CE	22	7112565	0.32329	1.49
DE	18.45	4510427	0.24448	1.97
ABC	22.42	10081744	0.44973	1.07
ABD	21.21	5645015	0.26609	1.81
ABE	16.48	5310025	0.32221	1.50
ACD	16.11	7960279	0.49414	0.98
ACE	16.69	6912595	0.41423	1.17
ADE	16.19	4350529	0.26868	1.80
BCD	31.48	16770866	0.53271	0.91
BCE	18.01	8618597	0.47861	1.01
BDE	17	4541733	0.26727	1.81
CDE	21.96	6314853	0.28762	1.68
ABCD	21.71	9130806	0.42067	1.15
ABCE	20.95	8419493	0.40185	1.20
ABDE	17.3	4395507	0.25405	1.90
ACDE	17.07	6171554	0.36149	1.34
BCDE	18.98	7674567	0.4044	1.19
ABCDE	20.01	5550942	0.27737	1.74

tends to consume more area since unrolling a loop tends to duplicate hardware components as clearly seen in the table. Also, the amount of speed up achieved using "loop unrolling" is approximately 1.53 over the base version. The amount of resources required by the "pipelining" method is much less compared to the "loop unrolling" strategy. However, the amount of speed up achieved using pipelining over the base version is almost negligible (i.e., 1.18 or 18% faster).

Table 6 shows the mean run times of the two optimization techniques used in the form of loop unrolling and loop pipelining. Also included in Table 6 are the run times of the base code (i.e., Scatter Search with no Impulse-C optimizations present) and the run time of Scatter Search in software.

4.5. Analysis of Results. To summarize, the previous results of our study show that directly converting Scatter Search into Handel-C does not result in a hardware implementation that runs faster than software. However, by carefully employing language-level transformations to improve the hardware that is synthesized, it is possible to produce hardware that runs marginally faster than software. The largest speedup (97%) was achieved when moving loop-invariant statements outside of the loops in which they appear, simplifying complex expressions into series of simpler computations and then performing the restructured and simplified statements in parallel with other statements. Moreover, there was little to no benefit in restructuring loops into more efficient forms, removing complex expressions from conditional tests, and simplifying deeply nested conditional statements. Although

Table 4: Handel-C: FPGA resource utilities for DE and base designs.

Resources	Implementation	
	base	DE
DSP48Es	1	1
Block RAM	13	13
Slice Reg	7369	7256
Slice LUTS	12,274	12,286
Slice LUT-FF	4214	4067
Maximum Freq.	15.36 MHz	18.45 MHz
Clock no.	8583039	4510427
Time	0.48278 Sec	0.24448 Sec

Table 5: Impulse-C (resource usage): a comparison among different Scatter Search implementations.

Resources	Implementation		
	Base	UnRoll	Pipeline
DSP48Es	51	792	97
Block RAM	121	124	125
Slice Reg	12231	169971	14281
Slice LUTS	21285	598407	24585
Slice LUT-FF	24535	679292	30052
Maximum Freq.	65.084 MHz	67.72 MHz	65.084 MHz
Clock no.	190388040	129010702	160753272
Time	2.925 Sec	1.905 Sec	2.4699 Sec

the results of the previous study are for Scatter Search, we believe that they will generalize to other population-based metaheuristics implemented using Handel-C, as these metaheuristics tend to have similar data, memory, and control structures. More generally, performance improvements with Handel-C occur iteratively, through an analysis of how the application is being compiled to hardware and through experimentation that Handel-C language programming allows.

5. Coarse-Grained Parallelization

Having determined that the performance improvements available in both Handel-C and Impulse-C through various fine-grained optimizations are minimal, we now seek to further improve the performance of Scatter Search through the use of data parallelism and pipelining. Data parallelism is a divide-and-conquer approach to parallelism where the data is divided and processed by multiple processing units running side by side in parallel. Pipelining, on the other hand, can be viewed as a form of parallelism where a process is divided into a set of stages, all of the stages run concurrently, and the output of one stage appears as the input to the next stage. The goal of pipelining is to increase system throughput by overlapping (in time) the execution of the stages. When seeking to apply these forms of parallelism to Scatter Search, we focus on three related methods: the subset-generation method, the combination method, and the improvement method. These related methods are targeted because they are all time-consuming bottlenecks, each having a run time complexity of $O(n^2)$. Moreover, all three methods exhibit characteristics that make them suitable for parallel implementation. The proposed parallel architecture is illustrated in Figure 2. In the subsections that follow, we elaborate on the contents of the architecture and explain how we parallelized the previous methods using a combination of data parallelism, pipelining, and a mix of both. Then, an implementation based on Handel-C is presented.

5.1. Parallelizing the Subset-Generation Method. The subset-generation method is a problem-independent method that is responsible for generating subsets of reference solutions that will be later used by the combination method to form new (candidate) solutions. In our implementation, we consider subsets consisting of all pair wise combinations of solutions in the reference set. As the reference set contains 10 solutions, a total of $(102 - 10)/2 = 45$ subsets of size two are generated. In the original (software) implementation employed by Laguna and Mart [3], these subsets were generated sequentially using a doubly nested loop and were stored in an array. However, we have chosen to unfold the loop in order to compute all 45 subsets in constant time. These subsets are then stored in a block RAM on the FPGA. The main reason for doing this is to allow the block RAM containing the subsets to be duplicated, as needed, when parallelizing the combination method, described below. The ability to duplicate data, thus making it available to multiple units when and where needed, is crucial for avoiding memory contention problems that would otherwise arise.

5.2. Parallelizing the Combination Method. The combination method is a problem-specific method that uses the subsets generated by the subset-generation method to generate new (candidate) solutions. The solutions in each subset are combined (in a problem-specific way) to produce one (or more) candidate solution. Our implementation follows from the implementation in [3] where two solutions are input and (linearly) combined to form a single, new solution. However, rather than using a single combination method to process all 45 subsets serially, we choose to parallelize the combination method by creating n identical combination methods. Each of the n combination methods is coupled to its own subset-generation method (implemented in block RAM) and is responsible for processing all of the subsets in each RAM. Note that the 45 subsets are evenly divided among the n subset methods (and their corresponding combination methods). For example, if $n = 5$, each of the five combination methods will obtain its input from exactly one of five block RAMs, and each block RAM will contain $45/5 = 9$ (unique) subsets of solutions. Each combination method will simultaneously read its assigned subsets (serially) from its own block RAM until all 9 subsets have been processed.

Table 6: Impulse-C results: comparison with Software Implementation.

Optimization version (with respect to base)	Clock freq. (MHz)	Average cycles	Average run time (s)	Speed up (with respect to software)
Software	—	—	0.48278	—
Impulse-C Base (No Opt)	65.084	190388040	2.92527	0.16503
Loop Unrolling	67.72	129010702	1.90506	0.25341
Loop Pipelining	65.084	160753272	2.46994	0.19546

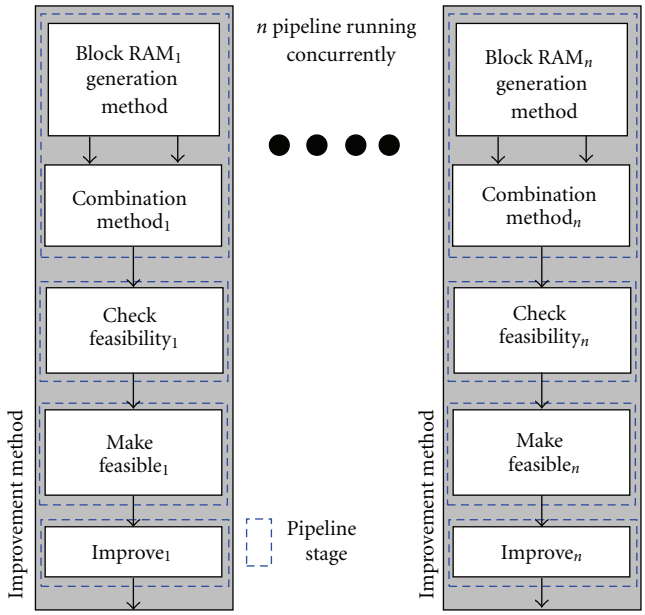

Figure 2: Multiple-pipeline architecture.

In addition to this type of data parallelism, each subset-generation and combination method pair acts as the first stage in a pipeline that includes the improvement method, as described next.

5.3. Parallelizing the Improvement Method. The improvement method is a problem-specific method that is responsible for improving the quality of the solutions produced by the combination method. However, as there is no guarantee that a solution provided by the combination method is feasible, the improvement method may have to repair (make feasible) the solution before seeking to improve its quality. In the original (software) implementation employed by Laguna and Mart [3], a two-phase improvement method is employed for the knapsack problem. The first phase is concerned with making infeasible solutions feasible. This is done by removing items from the knapsack until the capacity of the knapsack is no longer exceeded. Items are removed in order of their profit-to-weight (v_i/w_i) ratios, starting with the smallest. In phase two, feasible solutions are improved. This is done in a greedy fashion, by adding items back to the knapsack according to their profit-to-weight ratio, starting with the largest. The procedure stops once no more items can be added back to the knapsack without violating the capacity of the knapsack. We choose to implement the combination method, along with the subset generation and combination methods, as part of a 4-stage pipeline, and then replicate the pipeline.

The first stage of the pipeline consists of the subset-generation method and combination method, as described above, and is responsible for generating a stream of candidate solutions that appear as input to the improvement method. The improvement method itself is divided into three phases, each of which represents a single stage in the pipeline. Stage 2 of the pipeline checks to see if the solution from stage 1 is feasible. If infeasible, the solution is made feasible in stage 3. Clearly, this stage will sit idle if the original solution from stage 1 is found to be feasible. However, empirical evidence revealed that the best performance was achieved by implementing stages 2 and 3 as separate stages, rather than as a single stage. In general, having more pipeline stages helps to reduce deep logic and improve clock frequency. This is a direct result of the registers required between stages to hold data. These help to reduce the delay of otherwise critical paths in the circuit. Finally, stage 4 seeks to improve the feasible solution (provided by stage 3). Figure 3 shows the behavior of the pipeline over time. Once full, the pipeline concurrently combines two solutions to form a new candidate solution, checks the feasibility of a second candidate solution, repairs (makes feasible) a third candidate solution, and improves the quality of a fourth candidate solution. When implementing the stages in Handel-C, care was taken to balance the number of clock cycles in each stage to ensure good performance. Most importantly, as long as there is space on the FPGA the pipeline can be replicated to have multiple pipelines running in parallel, as illustrated in Figure 2.

5.4. Results of Parallelization. The architecture illustrated in Figure 2, and described in the previous subsections, was implemented in Handel-C using the tools described earlier. Care was taken to employ language-level optimizations DE, as discussed in Section 4.3, when implementing the architecture as these were identified as being the most helpful when seeking to improving run time. The parallel Scatter Search was tested using the same benchmarks and methods as described earlier.

Table 7 shows the speed up over software that can be achieved when using 1, 3, and 5 parallel pipelines, respectively. When just one pipeline is used, a speed up of approximately 23x is achieved. It is also worth mentioning that the

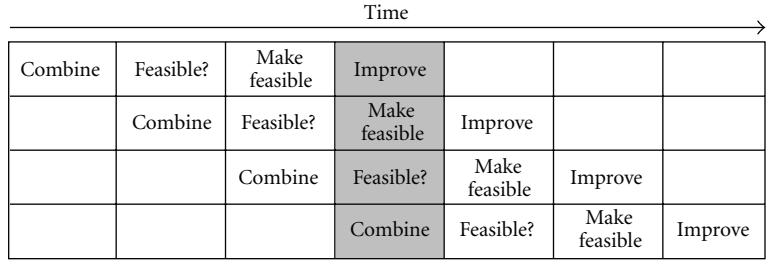

Figure 3: Single 4-stage pipeline.

Table 7: Speed up over software.

Pipeline units	Average hardware run time (s)	Speed up
1	0.02135	22.61
3	0.01738	27.78
5	0.02237	21.58

speed up achieved using "1" pipeline is almost 26x over the Handel-C base implementation without any optimization. This increases to 28x when 3 units are used but falls to 22x when five units are used. In the latter case, the decrease in speed up can be directly attributed to a reduction in clock frequency. When using 1 and 3 parallel pipelines, respectively, a clock frequency of 46 MHz and 47 MHz is achieved. However, when using 5 parallel pipelines the clock frequency falls to 35 MHz. Overall, all three hardware implementations achieve a significant performance improvement over software (with no statistical difference in solution quality).

6. Conclusion

The aim of this paper was twofold: (1) to perform an empirical study to determine the overall effectiveness of different statement-level optimizations (fine-grained parallelism) when using electronic system-level design in the form of Handel-C and Impulse-C to implement Scatter Search, and (2) to improve the run time performance of Scatter Search by parallelizing the algorithm and implementing it on an FPGA. The former was achieved by performing a full-factorial experiment to test interactions between different combinations of statement-level optimizations. Overall, this study showed that for a complex algorithm, like Scatter Search, only a marginal improvement in run time is possible when exploiting statement-level parallelism. Moreover, the study revealed that determining opportunities to exploit statement-level parallelism can be difficult. Coarse-grained parallelism, on the other hand, is better expressed at the system level. Through a combination of parallelism and pipelining, applied the subset-generation method, solution combination method, and improvement method, the resulting architecture, based on Handel-C, was able to achieve at least an order of magnitude speed up over software and as much as a 28x improvement in run time compared with software. Therefore, we conclude that Scatter Search is well suited to hardware acceleration.

For future work, we plan to investigate enhancements to the previous implementations, including parallelizing the reference set update, diversification, and improvement methods. We also plan to investigate further system-level implementations based on Impulse-C.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, Calif, USA, 1979.

[2] F. Glover, "A template for scatter search and path relinking," in *Artificial Evolution*, J. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, Eds., vol. 1363 of *Lecture Notes in Computer Science*, pp. 13–54, Springer, Berlin, Germany, 1998.

[3] M. Laguna and R. Mart, *Scatter Search: Methodology and Implementations in C*, Kluwer Academic, 2003.

[4] F. Garcia-Lopez, B. Melian-Batista, J. A. Moreno-Perez, and J. M. Moreno-Vega, "Parallelization of the scatter search for the p-median problem," *Parallel Computing*, vol. 29, no. 5, pp. 575–589, 2003.

[5] F. Garca-Lpez, M. G. Torres, B. Melin-Batista, J. A. M. Prez, and J. M. Moreno-Vega, "Parallel scatter search," in *Parallel Metaheuristics: A New Class of Algorithms*, pp. 223–244, John Wiley & Sons, 2005.

[6] B. Adenso-Diaz, S. Garcia-Carbajal, and S. Lozano, "An empirical investigation on parallelization strategies for scatter search," *European Journal of Operational Research*, vol. 169, no. 2, pp. 490–507, 2006.

[7] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "Implementation of a genetic algorithm on a Virtex-II pro FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, p. 287, 2009.

[8] S. Coe, S. Areibi, and M. Moussa, "A hardware memetic accelerator for VLSI circuit partitioning," *Computers and Electrical Engineering*, vol. 33, no. 4, pp. 233–248, 2007.

[9] A. Upegui, C. A. Pena-Reyes, and E. Sanchez, "An FPGA platform for on-line topology exploration of spiking neural networks," *Microprocessors and Microsystems*, vol. 29, no. 5, pp. 211–223, 2005.

[10] IEEE, "VASG: VHDL analysis and standardization group," 2008, http://www.vhdl.org/vasg/.

[11] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Springer, 2008.

[12] Agility, "Handel-C reference manual," 2009, http://www.mentor.com/products/fpga/handel-c/upload/handelc-reference.pdf.

[13] Impulse Accelerated Technologies, "Impluse c: Soft-ware tools for an accelerated world," 2010, http://www.impulseaccelerated.com/.

[14] Mentor Graphics, "Catapult C," 2010, http://www.mentor.com/products/esl/high -levelsynthesis/catapult-synthesis/.

[15] M. Walton, G. Grewal, and G. Darlington, "Parallel FPGA-based implementation of scatter search," in *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference*, pp. 1075–1082, ACM, New York, NY, USA, July 2010.

[16] M. Walton, G. Grewal, and G. Darlington, "Hardware acceleration of scatter search," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS '10)*, pp. 436–443, July 2010.

[17] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*, Springer, 2004.

[18] R. Kamat, V. Shelake, and S. Shinde, *Unleash the System On Chip Using FPGAs and Handel C*, Springer, 2010.

[19] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*, Prentice Hall, 2005.

[20] Mentor Graphics, "Handel-C synthesis methodology," 2010, http://www.mentor.com/products/fpga/handel-c/.

[21] Z. Michalewicz, *Genetic Algorithms Plus Data Structures Equals Evolution Programs*, Springer, New York, NY, USA, 1994.

[22] Xilinx, "ISE 11," 2010, http://www.xilinx.com/support/documentation/dt_ise11-1.htm.

[23] L. Garling and G. Woods, "Enhancing the analysis of variance (ANOVA) technique with graphical analysis and its application to wafer processing equipment," *IEEE Transactions on Components, Packaging, and Manufacturing Technology A*, vol. 17, no. 1, pp. 149–152, 1994.

[24] E. Alba, *Parallel Metaheuristics: a New Class of Algorithms*, Wiley-Interscience, 2005.

[25] R. O. Kuehl, *Design of Experiments: Statistical Principles of Research Design and Analysis*, Duxbury/Thomson Learning Press, Pacific Grove, Calif, USA, 2nd edition, 2000.